THE HELMHOLTZ ANALYTICS TOOLKIT (HEAT)
- A SCIENTIFIC BIG DATA LIBRARY FOR HPC -

19.09.2018 I **K. KRAJSEK**, C. COMITO, M. GÖTZ, B. HAGEMEIER, P. KNECHTGES, M. SIGGEL

EXTREME DATA WORKSHOP 2018

HEAT

- HeAT = **He**lmholtz **A**nalytics **T**oolkit
- Early alpha phase (project start: May 2018)
- Data Analytics framework for transparent distributed computation
- Build on PyTorch
- Developed in the open: *https://github.com/helmholtz-analytics* and *https://pypi.org/project/heat*
- Liberally licensed: MIT

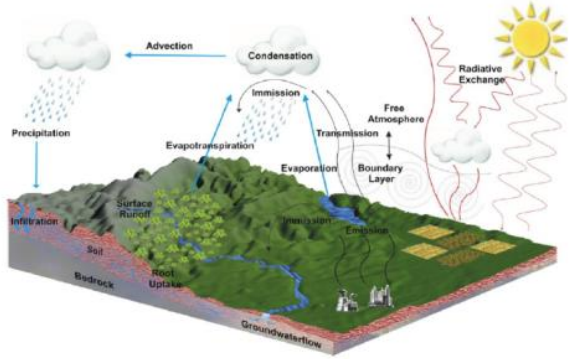
HELMHOLTZ ANALYTICS FRAMEWORK (HAF)

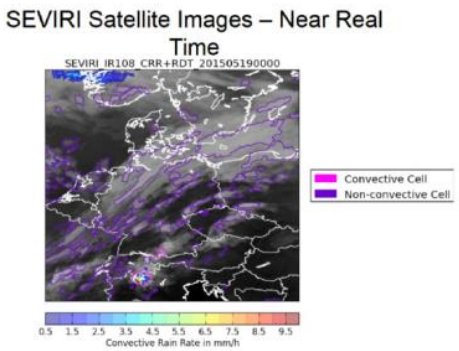
HAF – Goals and Objectives

Scientific big data analytics	- methodologies and tools for problems of highest data and compute complexity
Helmholtz Analytics Framework	- foster data science in Helmholtz - develop and exploit the Helmholtz Data Federation (HDF)
Use case driven co-design between	- domain scientists - data experts - infrastructure professionals
Create data analytics techniques	- in a systematic manner - domain-specific as well as generalizable and standardized

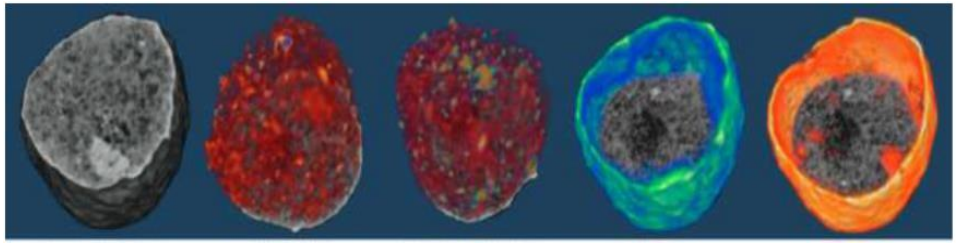
HAF USE CASES

Earth System Modelling

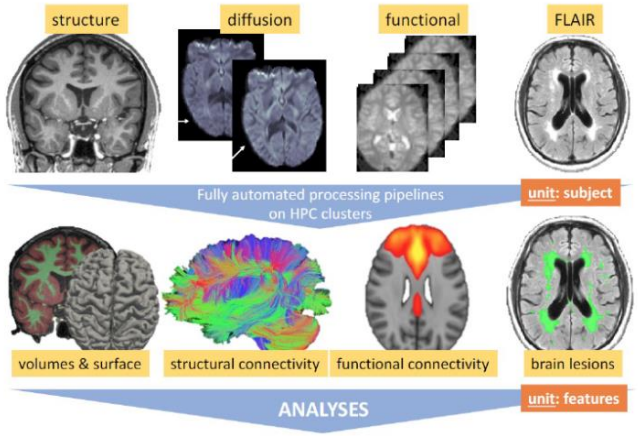




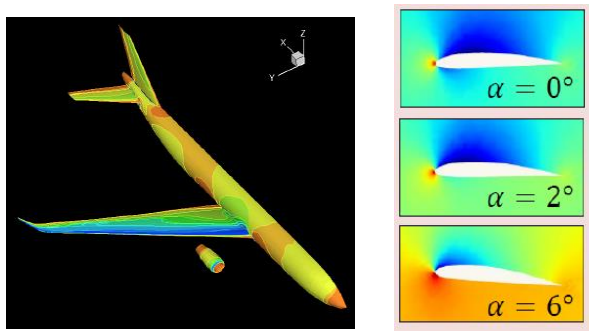
Research with Photons



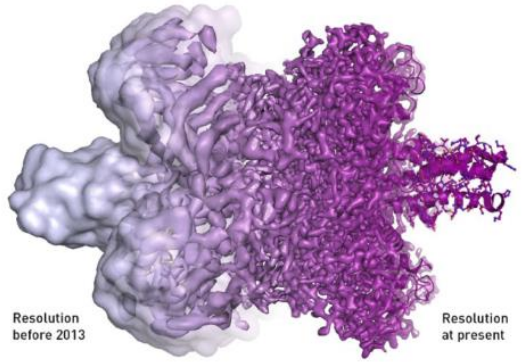
Neuroscience



Aeronautics and Aerodynamics



Structural Biology



HAF USE CASE METHODS

Clustering	kmeans (UC4, UC5, UC6, UC7,UC8), mean shift clustering (UC5)
Uncertainty quantification	Ensemble methods (UC1,UC2,UC3,UC4)
Dimension Reduction	autoencoder (UC5,UC7),reduced order models (UC7)
Feature learning	image descriptors (UC5), autoencoder (UC5,UC7)
Data Assimilation	Kalman filter (UC1), 4Dvar(UC2), particle filter/smoother(UC1, UC2)
Classifikation/Regression	Random forest, CNN (UC5), SVM (UC2,UC3,UC5), Lasso/Ridge regression(UC5)
Modelling	fiber tractography(UC5), point processes(UC6)
Optimizazion techniques	L-BFGS(UC5), simulated annealing(UC5)
Hyperparameter optimization	differential_evolution(UC5), grid search(UC5)
Interpolation	Radial basis function (UC7), Kriging(UC7)
Data Mining	Frequent item set mining(UC5)

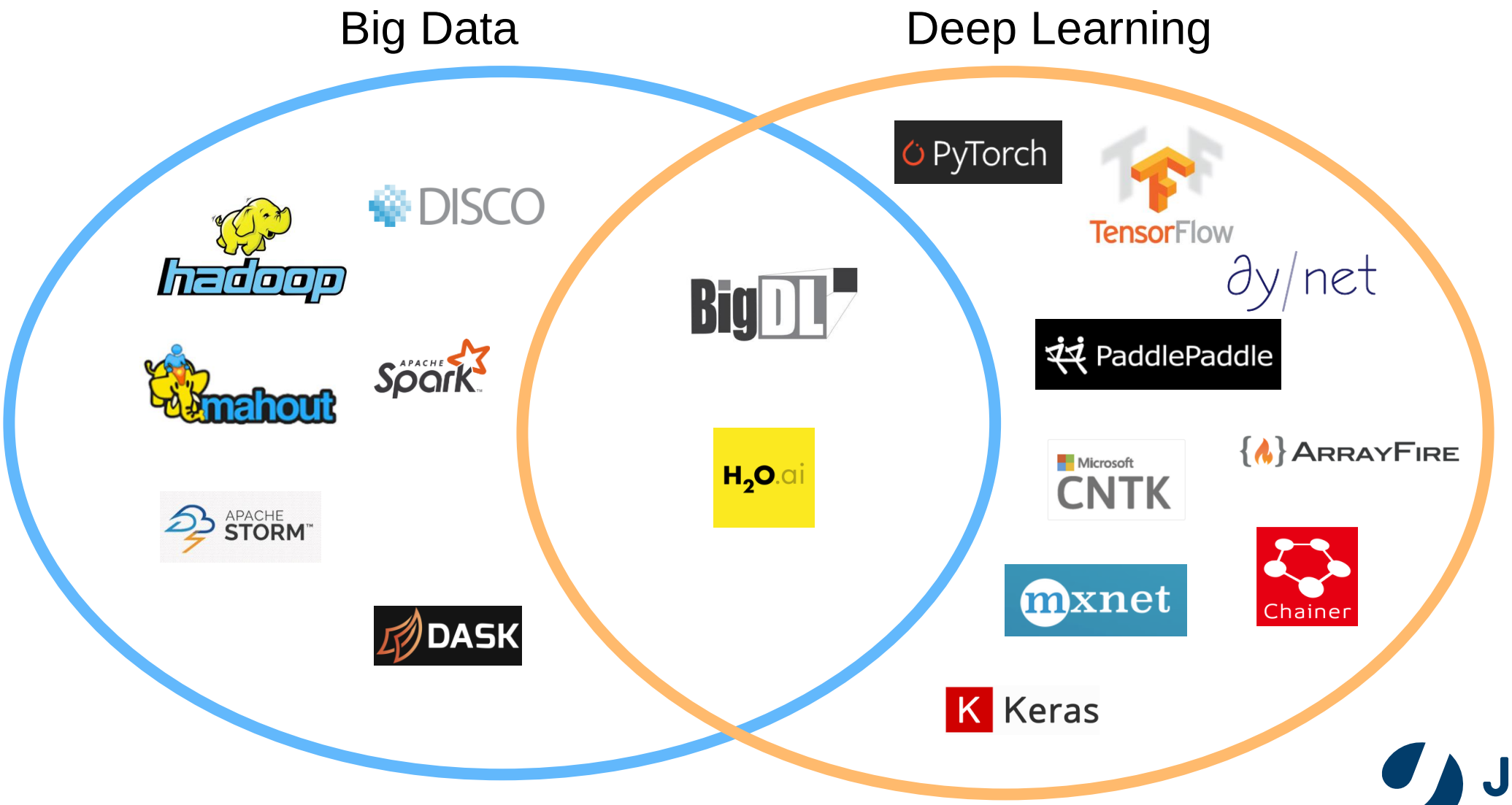
BIG DATA LANDSCAPE

V2 – Last updated 5/3/2017

© Matt Turck (@mattturck), Jim Hao (@jimrhao), & FirstMark (@firstmarkcap) mattturck.com/bigdata2017

FIRSTMARK EARLY STAGE VENTURE CAPITAL

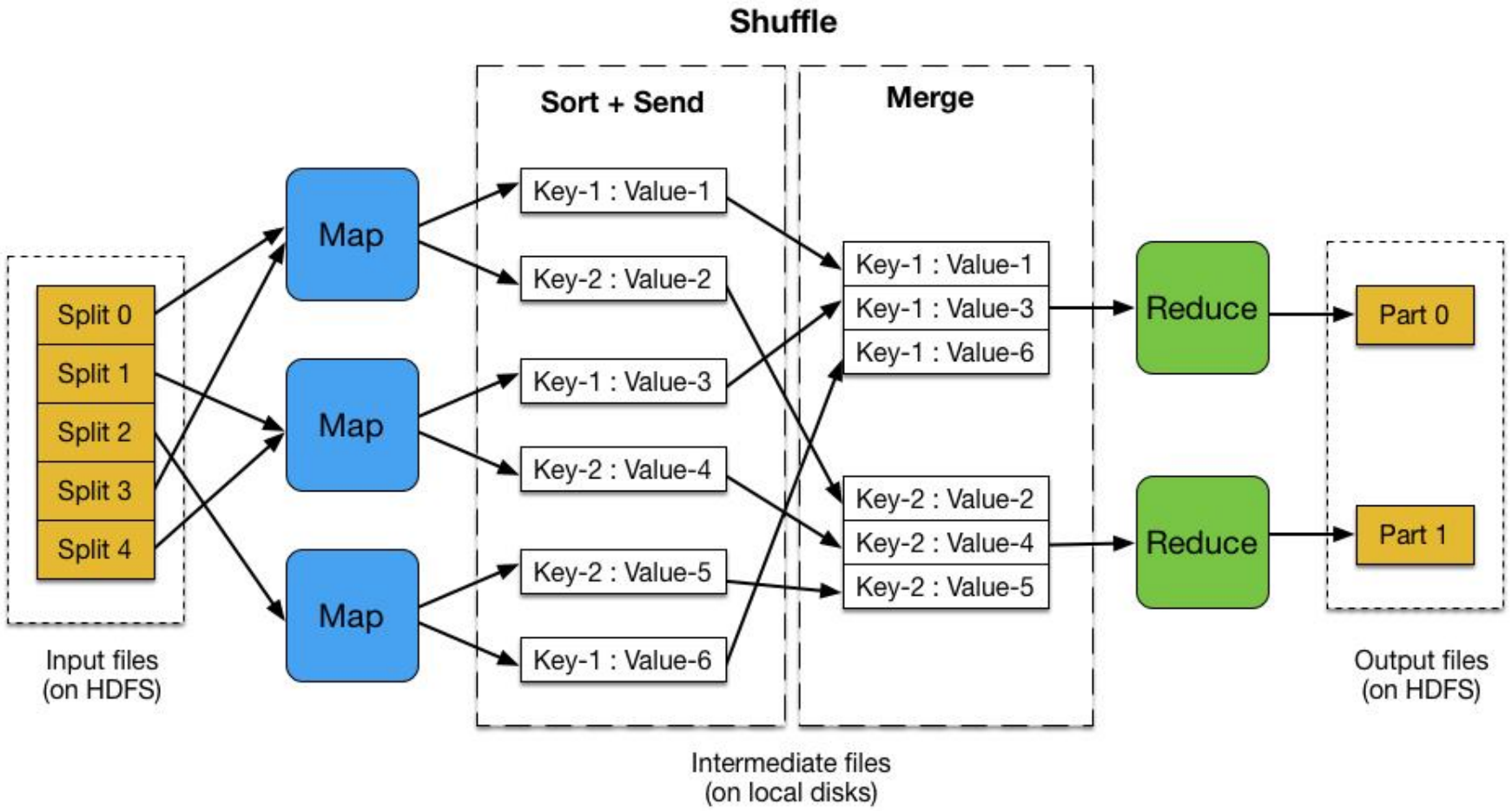
BIG DATA/DL LIBRARIES



MAP_REDUCE ALGORITHM

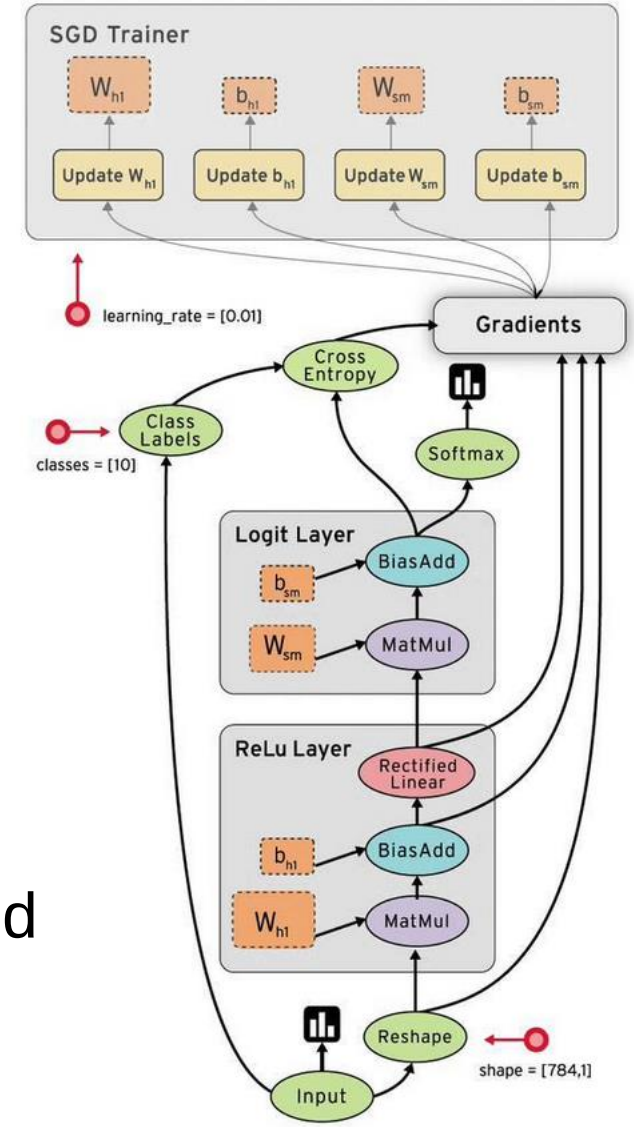
$$\mathrm{MAP} : K \times V \rightarrow (L \times W)^*$$

$$\mathrm{REDUCE} : L \times W^* \rightarrow X*$$



COMPUTATIONAL GRAPH

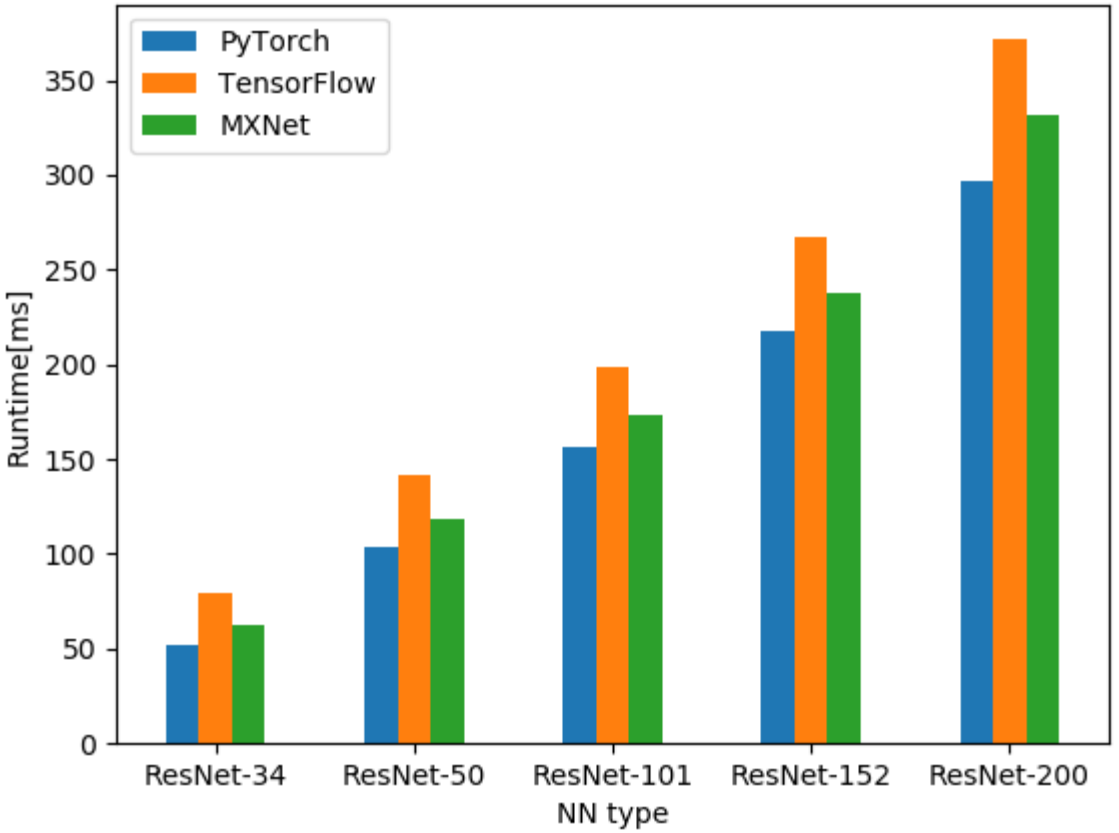
- Computations are defined as directed acyclic Graphs (DAGs)
- Graph is defined in high-level language (Python, C++, Go)
- Graph is compiled (e.g. CNTK) or built at run-time (e.g. PyTorch)
- Graph is executed on CPU/GPU
- Graph can be computed distributed but required manually assignments



BIG DATA LIBRARIES

	GPU	Distributed	MPI	AD	ML	Linear Algebra	ND-Tensors	Dynamic Graph
BigDL	■		■					■
Spark	■		■	■	■	■		■
Hadoop	■		■	■	■	■		■
Disco	■		■	■				■
H2O			■					■
Mahout			■	■				■
Intel Daal	■							■
Dask	■					■	■	■
TensorFlow								
PyTorch								
Arrayfire		■	■	■				■
MXNet								
PaddlePaddle								■
CNTK								■
DyNet		■	■					■
Chainer		■	■					

RUNTIME COMPARISION



Batch size of 32 x 224 x 224

ResNet architecture neural network

Run on single NVIDIA K80 GPU@JURECA

Similar results for other ML methods
(e.g. k-means)

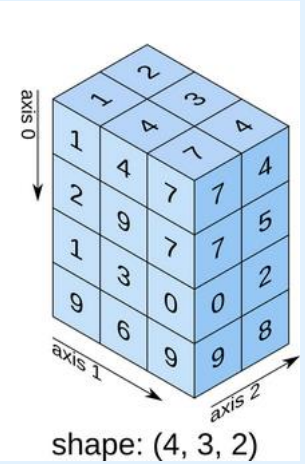
➡ Choose Pytorch as a base technology

NumPy

Runs on:

Data structure

ND-Tensor

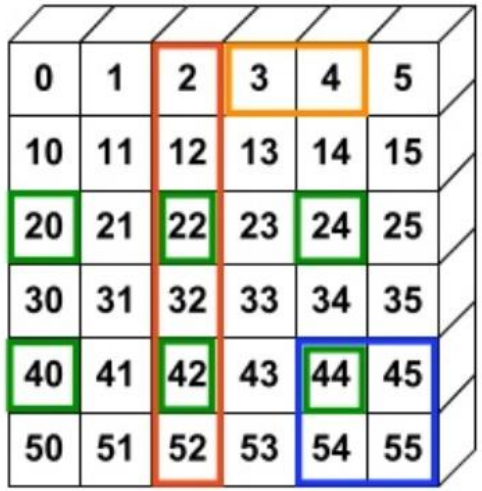


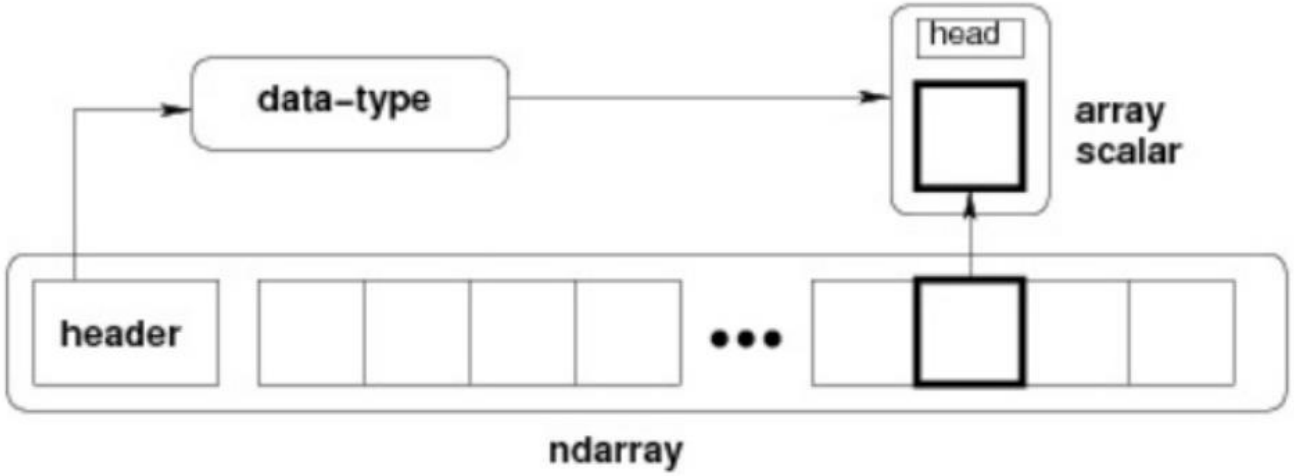
Operations

- Elementwise operations
- Slicing
- Matrix operations
- Reduction

NUMPY ARRAY

N dimensional homogeneous collection of scalars of the same data type





JÜLICH Forschungszentrum

ELEMTWISE OPERATIONS

```
a + b →  add(a,b)
a - b →  subtract(a,b)
a % b →  remainder(a,b)
```

```
a * b  →  multiply(a,b)
a / b  →  divide(a,b)
a ** b →  power(a,b)
```

Multiply by a scalar

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

Operator function

```
>>> add(a,b)
array([4, 6])
```

Element by element addition

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

In-place Operator function

```
# Overwrite contents of a.
# Saves array creation
# overhead.
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```

COMPARISON AND LOGICAL OPERATIONS

equal	(==)	not_equal	(!=)	greater	(>)
greater_equal	(>=)	less	(<)	less_equal	(<=)
logical_and		logical_or		logical_xor	
logical_not					

2-D Example

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])
# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```

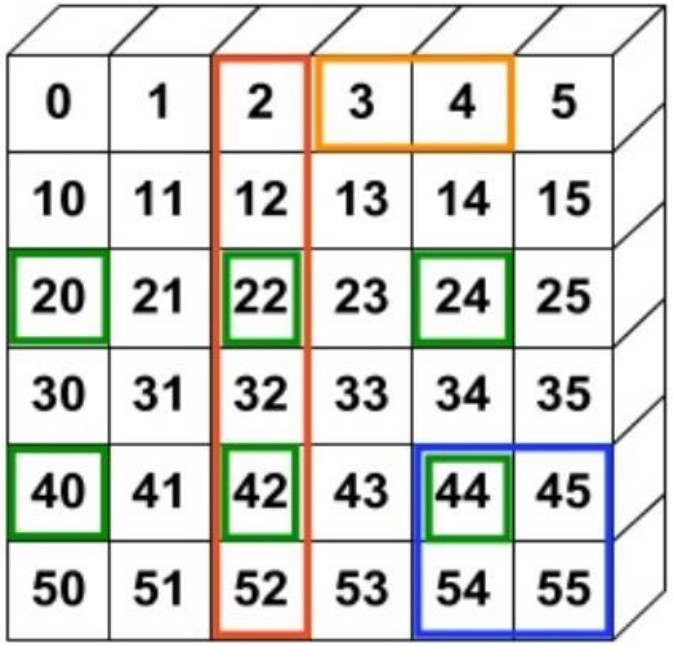
ARRAY SLICING

Slicing works much like standard Python slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```



Strides are also possible

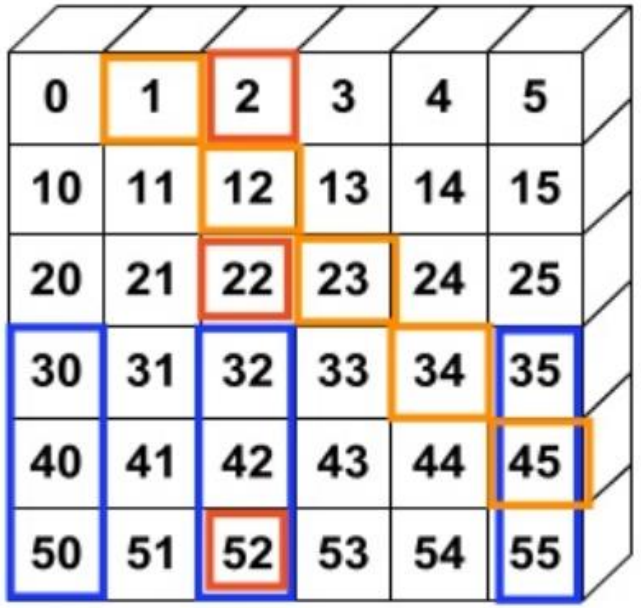
```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

FANCY INDEXING

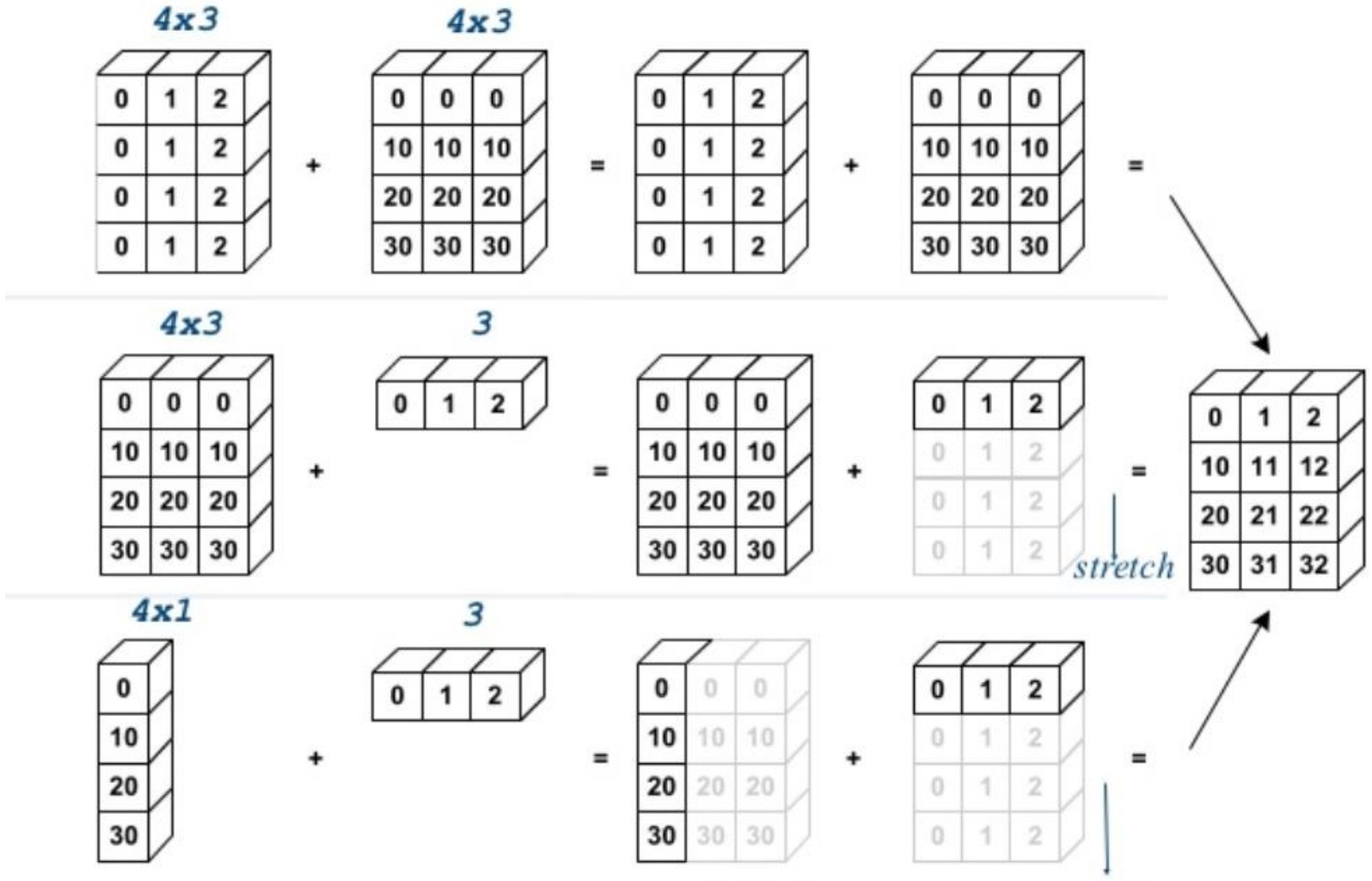
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



ARRAY BROACASTING



JÜLICH Forschungszentrum

REDUCTION METHODS

Sum Function

```
>>> a = array([[1,2,3],
               [4,5,6]], float)

# sum() defaults to adding up
# all the values in an array
>>> sum(a)
21.

# supply the keywork axis to
# sum along the 0th axis
>>> sum(a, axis=0)
array([5., 7., 9.])

# supply the keyword axis to
# sum along the last axis
>>> sum(a, axis=-1)
array([6. 15.])
```

Sum Array Method

```
# a.sum() defaults to adding
# up all values in an array
>>> a.sum()
21.

# supply an axis argument to
# sum along a specific axis
>>> a.sum(axis=0)
array([5., 7., 9.])
```

Product

```
# product along columns
>>> a.prod(axis=0)
array([4., 10., 18.])

# functional form
>>> prod(a, axis=0)
array([4., 10., 18.])
```

MATRIX OPERATIONS

Matrix Multiplication

```
>>> a = [[1, 0],
         [0, 1]]
>>> b = [[4, 1],
         [2, 2]]

# Matrix product of a and b
>>> np.matmul(a, b)
array([[4, 1],
       [2, 2]])
```

Determinant

```
>>> a = np.array([[1, 2],
                  [3, 4]])

# compute determinant
>>> np.linalg.det(a)
-2.0
```

Eigenvalue decomposition

```
>>> w, v = np.linalg.eig(np.diag((1, 2, 3)))
>>> w; v
array([ 1.,  2.,  3.])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Matrix inversion

```
a = np.array([[1., 2.],
              [3., 4.]])

>>> ainv = np.linalg.inv(np.matrix(a))
>>> ainv
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

PYTORCH CONCEPT

What is PyTorch?

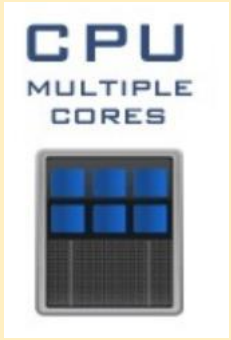
Python based scientific computing package targeted at two sets of audiences:

- A replacement for **NumPy** to use the power of **GPUs**
- A **deep learning** research platform that provides maximum flexibility and speed

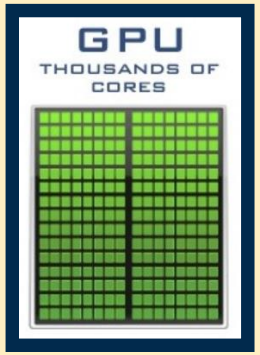
Automated differentation (DA)

PyTorch

Runs on:

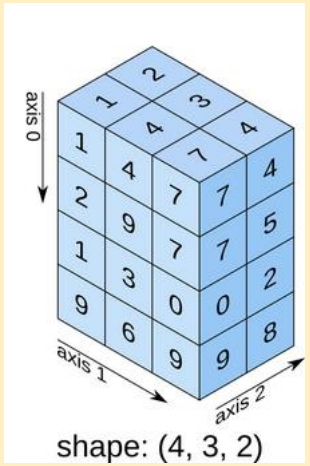


or



Data structure

ND-Tensor



Operations

- Elementwise operations
- Slicing
- Matrix operations
- Reduction
- **Automatic differentation**

PYTOCH: RUN ON GPU

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)                       # or just use strings ``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```

PYTOCH: AUTOMATIC DIFFERENTATION

How can we do the same as above with PyTorch's `autograd` package?

First, it should be obvious that we have to represent our original function in Python as such:

```
y = 5*x**4 + 3*x**3 + 7*x**2 + 9*x - 5
```

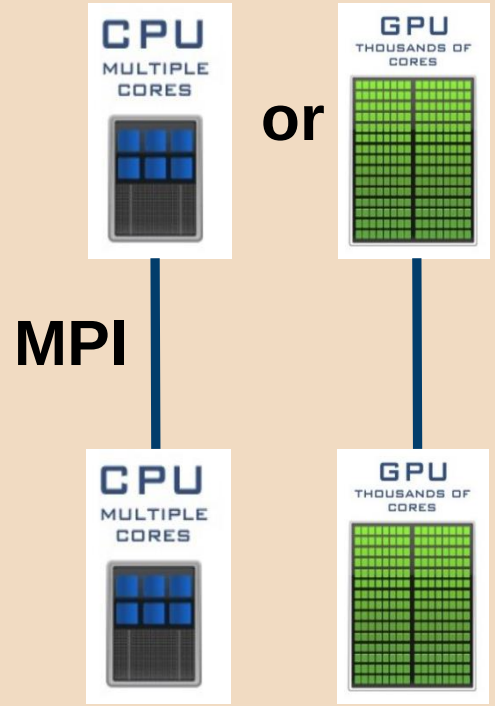
```
import torch

x = torch.autograd.Variable(torch.Tensor([2]),requires_grad=True)
y = 5*x**4 + 3*x**3 + 7*x**2 + 9*x - 5

y.backward()
x.grad
```

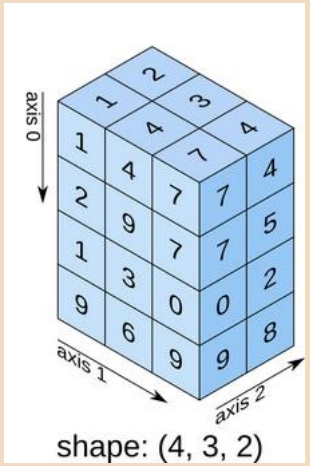
HeAT

Runs on:



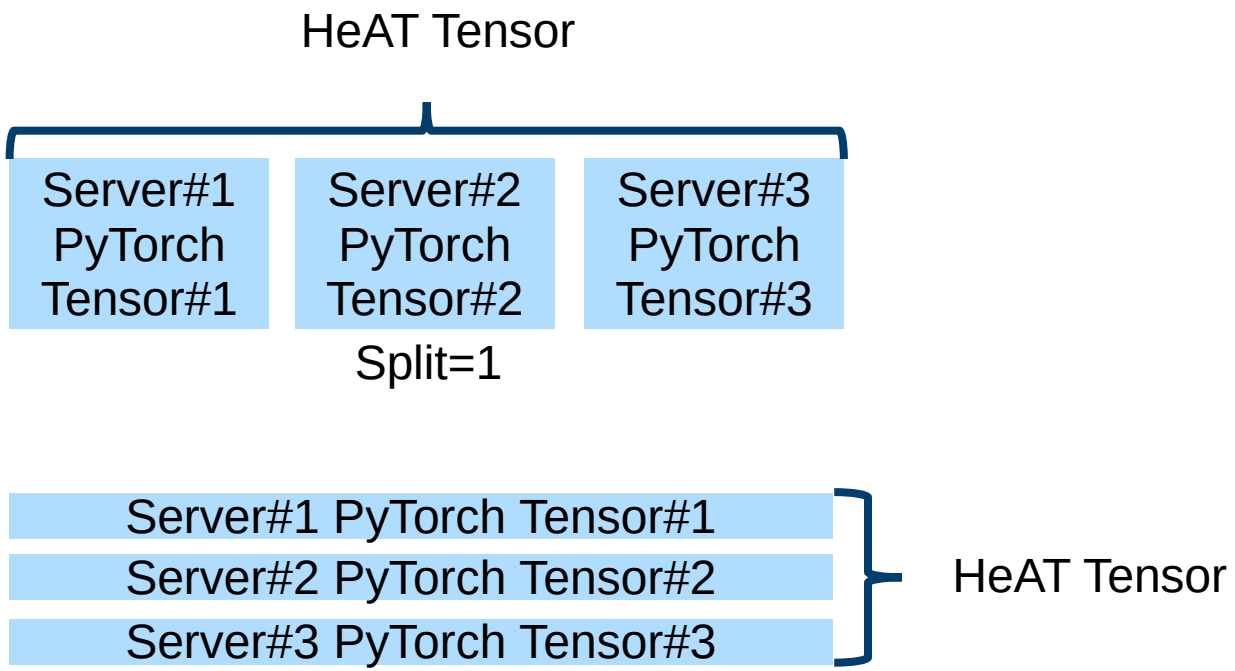
Data structure

ND-Tensor



Operations

- Elementwise operations
- Slicing
- Matrix operations
- Reduction
- Automatic differentation



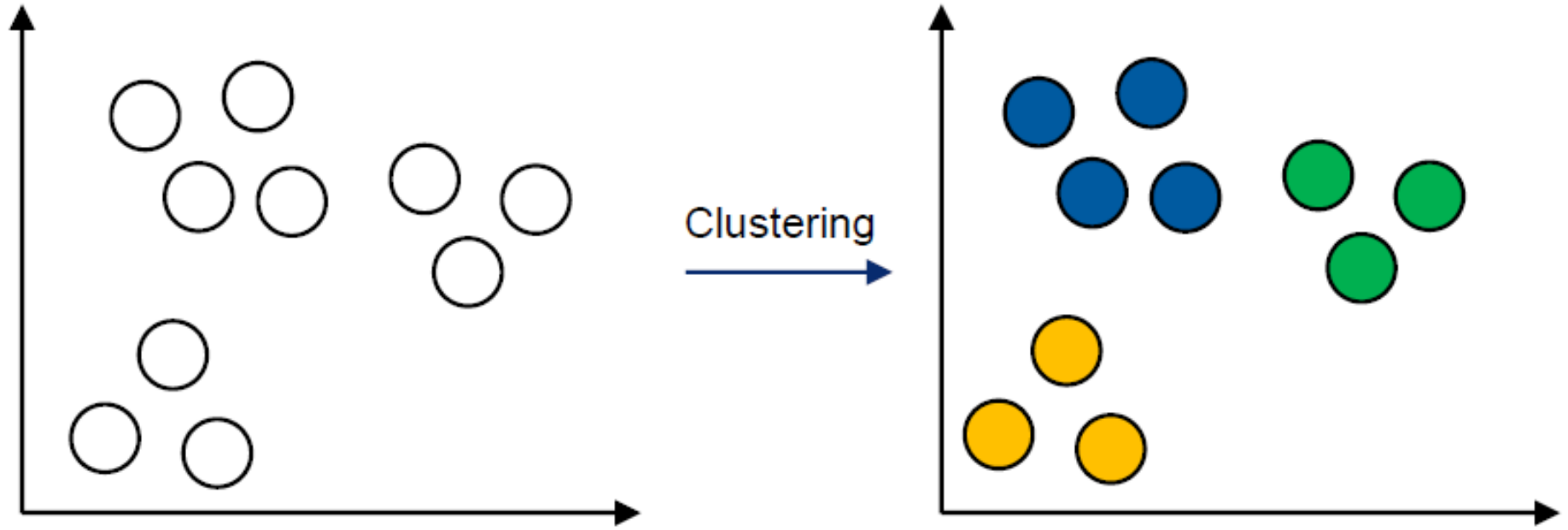
Example:

```
import heat as ht
# construct a range tensor
>>> range_data = ht.arange(6, split=1)
```

Server#1	Server#2	Server#3
[0, 1]	[2, 3]	[4, 5]

```
>>> range_data.mean()
2.5
>>> range_data.argmax()
5
```

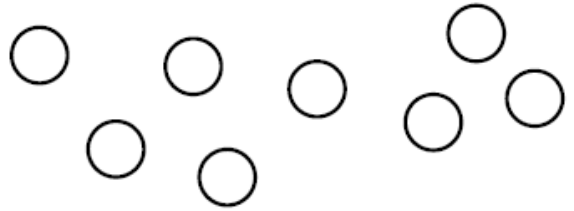
EXAMPLE: CLUSTERING



EXAMPLE: K-MEANS

KK:This has to be updated!!!!!

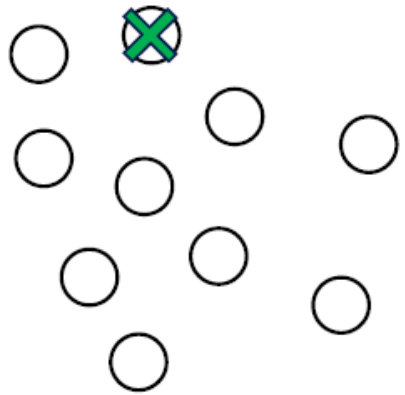
- Core idea: ***k* clusters** around centroids



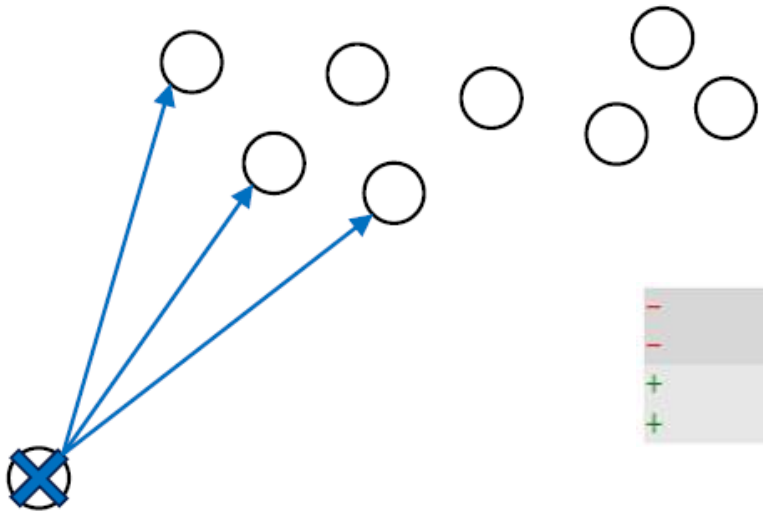
- Iterative minimization
  - $\arg\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \bar{x}\|^2$
  - Other matrics possible

```
-       np.random.seed(seed)
-       return np.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))
+       ht.random.set_gseed(seed)
+       return ht.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))
```

- Algorithm sketch
  - Choose *k* centroids
  - For each points calculate distance to centroids
  - Assign point to **closest centroid**
  - Estimate new centroids as **mean** of points
  - Repeat until **convergence**

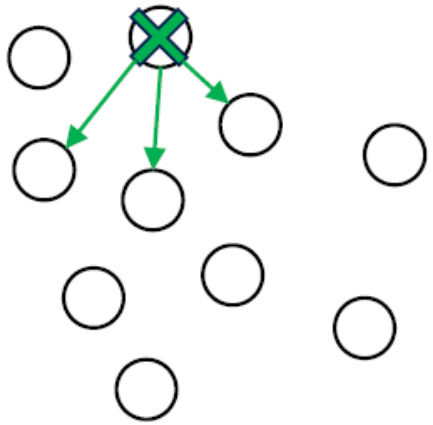


EXAMPLE: K-MEANS

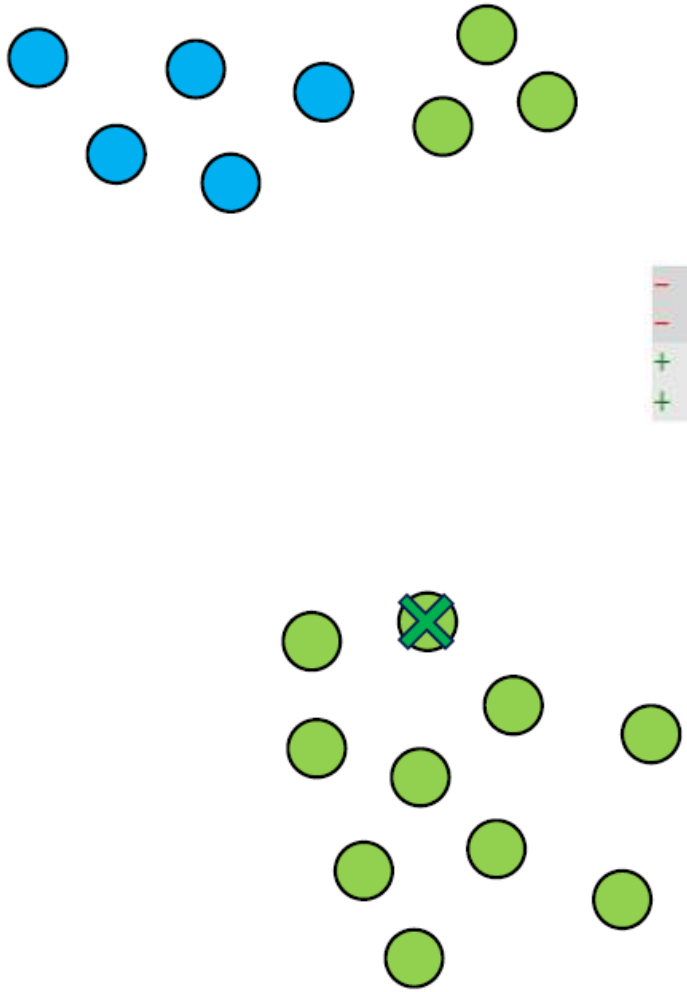


Numpy vs. HeAT

```
-           distances = ((data - centroids) ** 2).sum(axis=1,keepdims=True)
-           matching_centroids = np.expand_dims(distances.argmin(axis=2),axis=2)
+           distances = ((data - centroids) ** 2).sum(axis=1)
+           matching_centroids = distances.argmin(axis=2)
```



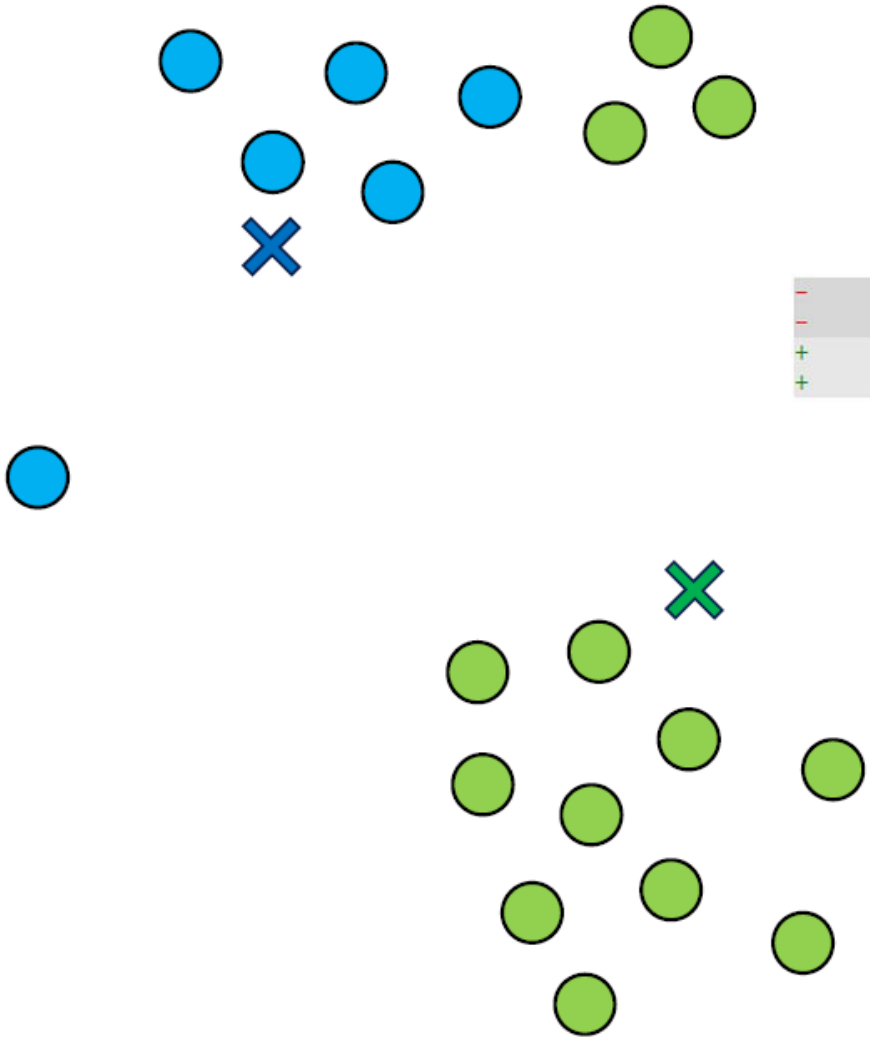
EXAMPLE: K-MEANS



Numpy vs. HeAT

```
-           distances = ((data - centroids) ** 2).sum(axis=1,keepdims=True)
-           matching_centroids = np.expand_dims(distances.argmin(axis=2),axis=2)
+           distances = ((data - centroids) ** 2).sum(axis=1)
+           matching_centroids = distances.argmin(axis=2)
```

EXAMPLE: K-MEANS



Numpy vs. HeAT

```
-               selection = (matching_centroids == i).astype(np.int64)
-               new_centroids[:, :, i:i + 1] = ((data * selection).sum(axis=0,keepdims=True) /
+               selection = (matching_centroids == i).astype(ht.int64)
+               new_centroids[:, :, i:i + 1] = ((data * selection).sum(axis=0) /
                                                selection.sum(axis=0).clip(1.0, sys.maxsize))
```

NUMPY VS. HEAT

Important from a users perspective:

- Aiming for numpy-compatibility
- Ideally, you just need to replace np by ht

```
@@ -1,9 +1,9 @@
 import sys

-import numpy as np
+import heat as ht


-class KMeansNP:
+class KMeans:
     def __init__(self, n_clusters, max_iter=1000, tol=1e-4, random_state=42):
         # TODO: document me
         # TODO: extend the parameter list
@@ -17,12 +17,12 @@
         # TODO: document me
         # TODO: extend me with further initialization methods
         # zero-centered uniform random distribution in [-1, 1]
-        np.random.seed(seed)
-        return np.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))
+        ht.random.set_gseed(seed)
+        return ht.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))

     def fit(self, data):
         # TODO: document me
-        data = np.expand_dims(data,axis=2)
+        data = data.expand_dims(axis=2)

         # initialize the centroids randomly
         centroids = self.initialize_centroids(self.n_clusters, data.shape[1], self.random_state)
@@ -30,13 +30,13 @@

         for epoch in range(self.max_iter):
             # calculate the distance matrix and determine the closest centroid
-            distances = ((data - centroids) ** 2).sum(axis=1,keepdims=True)
-            matching_centroids = np.expand_dims(distances.argmin(axis=2),axis=2)
+            distances = ((data - centroids) ** 2).sum(axis=1)
+            matching_centroids = distances.argmin(axis=2)

             # update the centroids
             for i in range(self.n_clusters):
-                selection = (matching_centroids == i).astype(np.int64)
-                new_centroids[:, :, i:i + 1] = ((data * selection).sum(axis=0,keepdims=True) /
+                selection = (matching_centroids == i).astype(ht.int64)
+                new_centroids[:, :, i:i + 1] = ((data * selection).sum(axis=0) /
                                                 selection.sum(axis=0).clip(1.0, sys.maxsize))
```

SUMMARY

BLABLA